Backend Frameworks

Backend frameworks are a set of tools and libraries used to simplify and accelerate application development. They provide functions for handling HTTP requests, routing, database operations, and more. Common Python backend frameworks include Django, Flask, and FastAPI. For example, Django is a high-level Python framework, while Flask is a lightweight framework that is simpler to use.

Django Framework

Django consists of several components:

- **Model:** Defines the data structure and database tables.
- Example: **Suppose we are developing a voting application, we need a model to represent a Question and another to represent a Choice.**

- **View:** Handles HTTP requests and returns responses.
- Example: **Create a view to display the homepage of the voting application.**

- **Template:** Defines the HTML page structure to display data.
- Example: **Create a simple template to display a list of questions.**

- URL Configuration: Defines URL routing to direct requests to the appropriate views.
- Example**: Configure a URL to access the homepage of the voting application.**

Database Integration with Django

First, Django supports multiple databases and connects to them via the `settings.py` file. Django supports various databases like SQLite, MySQL, PostgreSQL, etc. To connect Django with a database, we need to configure the `settings.py` file in the Django project.

Migrations

- Migrations: Migrations are Django's mechanism **for syncing the model with the database schema.**
- When the model changes, Django generates migration files that record the database schema changes.
- By running migration commands, Django updates the database schema according to the migration files.

Django REST Framework and Object-Oriented Concepts

Object-Oriented Programming (OOP) abstracts real-world entities into objects, solving problems through their interactions. Core concepts include classes, objects, attributes, and methods.

- **Class: An abstract description of a category of objects,** defining common attributes and behaviors.
- **Object: A specific instance of a class,** with its own attributes and behaviors.
- **Attributes: Characteristics or states of an object.**
- **Methods: Actions or behaviors that an object can perform.**

Example:

Class: Car

Attributes: Color, brand, model, engine type

Methods: Start, accelerate, brake, stop

Object: A red Tesla Model S

Attributes: Color is red, brand is Tesla, model is Model S, engine type is electric

Methods: This car can start, accelerate, brake, and stop

Class: Book

Attributes: Author, title, published_date …

Methods: Write, read, borrow, purchase

Subclass: Author

Attributes: Name, birth date, gender

Specific Object

- Attributes: Author: J.K. Rowling
- Title: Harry Potter and the Philosopher's Stone
- Published Date: 1997-06-26

For the class of this object, we can see the specific object. For example, a book called Harry Potter and the Sorcerer's Stone is object-oriented, so it can also be applied to our sensors. Make an association, such as the various properties of sensors, such as the various properties of measurements.

Define the Model

To implement OOP in Django, we start by defining the models. **Below is an example of defining two models: Book and Author. The Book model includes the title, author, and published_date attributes, while the Author model includes the name attribute.**

```
from django.db import models

class Book(models.Model): # Define a model class named Book to represent books
    title = models.CharField(max_length=200) # Define the book title, using
CharField for a character field with a maximum length of 200

    author = models.ForeignKey('Author', on_delete=models.CASCADE) # Define the
book author, using ForeignKey to link to the Author model
    published_date = models.DateField() # Define the publication date, using
DateField for a date field

    # Define the __str__ method to return the string representation of the object
    # This makes it easier to view objects in the admin interface and shell
    def __str__(self):
        return self.title

class Author(models.Model): # Define a model class named Author to represent
authors
    name = models.CharField(max_length=100) # Define the author's name, using
CharField for a character field with a maximum length of 100

    # Define the __str__ method to return the string representation of the object
    def __str__(self):
        return self.name
```

Define a Serializer: Serializers convert model instances to other formats.

```
from rest_framework import serializers # Import the serializers module from
Django REST framework
from .models import Author, Book # Import the Author and Book models from the
previously created models module
```

```
class BookSerializer(serializers.ModelSerializer): # Define the BookSerializer
class, inheriting from serializers.ModelSerializer
    class Meta: # Define the Meta inner class to specify serializer metadata
        model = Book # Specify that the serializer corresponds to the Book model
        fields = ['id','title','author','published_date'] # Specify the list of
fields to include in the serializer

class AuthorSerializer(serializers.ModelSerializer):
    class Meta:
        model = Author # Specify that the serializer corresponds to the Author
model
        fields = ['id','name']
```

This code defines two serializer classes: BookSerializer and AuthorSerializer, **for converting Book and Author model instances to formats like JSON and vice versa. The fields to be included in the serialization are specified.**

- BookSerializer **contains id, title, author and published_date fields.**
- AuthorSerializer **contains id and name fields.**

This step has a fixed format, basically writing class ...Serializer. Then inside is model = class itself, fields = id plus other attributes.

Define a ViewSet for CRUD Operations

```
from rest_framework import viewsets # Import the viewsets module from Django REST
framework
from .models import Author, Book # Import the Author and Book models from the
current application's models module
from .serializers import AuthorSerializer, BookSerializer # Import the
AuthorSerializer and BookSerializer from the current application's serializers
module

class BookViewSet(viewsets.ModelViewSet): # Define a viewset class named
BookViewSet, inheriting from viewsets.ModelViewSet
    queryset = Book.objects.all() # Define the dataset to query for the viewset
    serializer_class = BookSerializer # Specify the serializer class to use for
the viewset

class AuthorViewSet(viewsets.ModelViewSet): # Define a viewset class named
AuthorViewSet, inheriting from viewsets.ModelViewSet
    queryset = Author.objects.all()
    serializer_class = AuthorSerializer
```

This code defines two view sets: **BookViewSet and AuthorViewSet, which are used to handle API requests for Book and Author models.**

BookViewSet inherits from viewsets.ModelViewSet and provides a default implementation for CRUD operations on the Book model. It uses Book.objects.all() to query all Book objects as a data set, and uses BookSerializer for serialization and deserialization. (The same applies to AuthorViewSet)

This part is to satisfy all request queries. So the operation done here is to query all books, so objects.all() returns all books. Finally, there is a fixed format serializer_class = BookSerializer / serializer_class = AuthorSerializer. If you want to do other operations, such as querying a specific book, you can also write a corresponding function to complete it in this part.

Define the Router

```
from rest_framework.routers import DefaultRouter # Import the DefaultRouter
module from Django REST framework
from .views import AuthorViewSet, BookViewSet # Import the AuthorViewSet and
BookViewSet from the current application's views module

router = DefaultRouter() # Create an instance of DefaultRouter
router.register(r'authors', AuthorViewSet) # Register AuthorViewSet with the
router, using 'authors' as the URL prefix
router.register(r'books', BookViewSet) # Register BookViewSet with the router,
using 'books' as the URL prefix

urlpatterns = router.urls # Set urlpatterns to include all URLs automatically
generated by the router
```

This code is used to set up the URL routing of Django REST framework. It automatically generates URLs for handling requests related to the Author and Book models by creating a DefaultRouter instance and registering the view set. The specific steps are as follows:

- Import the necessary modules and view set classes.
- Create a DefaultRouter instance.
- Register AuthorViewSet and BookViewSet in the router, using authors and books as URL prefixes respectively.
- Assign the URLs generated by the router to urlpatterns so that Django can use these URLs to handle requests.

This part follows a fixed format. Simply import the viewsets created above and then use router.register(), which is equivalent to subscribing or registering the viewsets. This URL becomes the address of the route.

The Entire Workflow

- Define the Models: **Define the desired models, such as books or sensors.**
- Define the Serializer: **Convert model instances to and from formats like JSON.**
- Create ViewSets: **Handle operations like querying specific sensors or returning all sensors.**
- Register ViewSets with URLs: **Make the viewsets accessible via URLs.**